

FormCalc 7.5

S. Agrawal

LNM IIT, Rupa Ki Nagal, Post-Sumel, Via Jamdoli, Jaipur-302031, India

E-mail: shivam@mpp.mpg.de

T. Hahn*

MPI für Physik, Föhringer Ring 6, D-80805 Munich, Germany

E-mail: hahn@mpp.mpg.de

E. Mirabella

MPI für Physik, Föhringer Ring 6, D-80805 Munich, Germany

E-mail: mirabell@mpp.mpg.de

We present additions and improvements in Version 7.5 of FormCalc, most notably OPP methods, Output in C, MSSM initialization via FeynHiggs, and Analytic tensor reduction, as well as a parallelized Cuba library for numerical integration.

Report MPP-2012-136

*Loops and Legs in Quantum Field Theory – 11th DESY Workshop on Elementary Particle Physics,
April 15–20, 2012
Wernigerode, Germany*

*Speaker.

1. Introduction

The Mathematica package FormCalc [1] simplifies Feynman diagrams up to one-loop order generated by FeynArts [2]. It provides both the analytical results and can generate code for the numerical evaluation of the squared matrix element.

Mathematica's powerful language enables users to easily inspect and modify results and should be considered a feature, not the deficiency other packages claim vindicates their use of e.g. Python.

This note presents improvements and additions in FormCalc 7.5 and the numerical integration package Cuba 3 [3], which is also included in FormCalc:

- Unitarity methods (OPP),
- Parallelization of helicity loop,
- Output in C and Improved code generation,
- Command-line parameters for model initialization,
- MSSM initialization via FeynHiggs,
- Analytic tensor reduction,
- Auxiliary functions for operator matching,
- Built-in parallelization in Cuba.

2. Unitarity methods (OPP)

FormCalc 7 can generate code which uses the OPP (Ossola, Papadopoulos, Pittau [4]) unitarity methods as implemented in the two libraries CutTools [5] and Samurai [6]. Rather than introducing Passarino–Veltman (PV [7]) tensor coefficient functions, the entire numerator is placed in a subroutine, as in:

$$\varepsilon_1^\mu \varepsilon_2^\nu B_{\mu\nu}(p, m_1^2, m_2^2) = B_{\text{cut}}(2, N, p, m_1^2, m_2^2), \quad \text{where} \quad N(q_\mu) = (\varepsilon_1 \cdot q) (\varepsilon_2 \cdot q).$$

The numerator subroutine N will be sampled by the OPP function (B_{cut} in this example). The first argument of B_{cut} , 2, gives the maximum power of the integration momentum q in N .

Subexpressions of the numerator function (coefficients, summands, etc.) which do not depend on q are pulled out and computed once, ahead of invoking the OPP function, using FormCalc's abbreviating machinery [8]. In particular in BSM theories, these coefficients can be lengthy such that pulling them out significantly increases performance.

The CutTools and Samurai libraries have minor differences in calling conventions but are otherwise similar enough to let the preprocessor handle the switching. That is, one does not need to re-generate the Fortran code in order to link with the other library. Specifically, the following steps must be taken in order to use the OPP method in FormCalc:

- The amplitudes must be prepared with `CalcFeynAmp[. . . , OPP → n]` ($n < 100$).

- In the generated code, the OPP library (CutTools or Samurai) must be chosen and the declarations in `opp.h` be included. This is most conveniently done in `user.h`, in the following structure:

```
#ifndef USER_H
#define USER_H
* declarations for the whole file (e.g. preprocessor defs)
#define SAMURAI                (or CUTTOOLS)
#else
* declarations for every subroutine
#include "opp.h"                (necessary for OPP)
#include "model_sm.h"
#endif
```

We have presently compared a handful of $2 \rightarrow 2$ and $2 \rightarrow 3$ scattering reactions, both QCD and electroweak, and found agreement to about 10 digits between PV and OPP, with CutTools and Samurai delivering results of similar quality. This shows that the method is working.

The performance is lagging quite a bit, however, at least when applying the OPP method naively and for lower-leg multiplicities. The principal difference is that the numerator function imposes a helicity dependence on the OPP function such that, unlike the PV tensor coefficients, it cannot be hoisted out of the helicity loop.

The following improvements have been made to optimize performance:

- Our implementation admits mixing PV decomposition with OPP in the sense that one chooses an integer n starting from which an n -point function is treated with OPP methods. For example, $\text{OPP} \rightarrow 4$ means that A, B, C functions are treated with PV and D and up with OPP. A negative n indicates that the rational terms for the OPP integrals shall be added analytically whereas else their computation is left to the OPP package.
- The number of OPP calls turns out to be more detrimental to performance than the complexity of the numerators. The simplification strategy for OPP integrals is thus to join, rather than split (as for PV), denominators. A loop integral whose denominators form a complete subset of another are joined with the latter, as in

$$\frac{N_4}{D_0 D_1 D_2 D_3} + \frac{N_3}{D_0 D_1 D_2} \rightarrow \frac{N_4 + D_3 N_3}{D_0 D_1 D_2 D_3}$$

Furthermore, simplifications that break up loop integrals are suppressed, such as the cancellation of q^2 -terms, e.g.

$$\frac{q^2}{(q^2 - m^2) D_1 D_2} \not\rightarrow \frac{1}{D_1 D_2} + \frac{m^2}{(q^2 - m^2) D_1 D_2}.$$

- Profiling the code pointed us to inefficiencies in the evaluation of fermion chains. In older FormCalc versions, the computation proceeded through nested invocations of elementary operations (2-component matrix–vector and vector–vector products [9]). Inlining these functions in a portable way in Fortran was syntactically not straightforward, so we switched to

single function calls for an entire chain, as in:

$$\begin{aligned} \langle u | \sigma_\mu \bar{\sigma}_\nu \sigma_\rho | v \rangle k_1^\mu k_2^\nu k_3^\rho &=: \langle u | k_1 \bar{k}_2 k_3 | v \rangle \\ \text{old} &= \text{SxS}(u, \text{VxS}(k_1, \text{BxS}(k_2, \text{VxS}(k_3, v)))) \\ \text{new} &= \text{ChainV3}(u, k_1, k_2, k_3, v) \end{aligned}$$

As the profiler no longer ‘sees’ these now-inlined functions, we cannot quote a concrete figure for the performance gain; a naive before–after comparison of wall-clock time indicates an improvement on the order of 5–10%, however.

- To avoid the evaluation of integrals whose prefactor is known to be exactly zero from helicity considerations we added an “helicity delta” argument to each OPP integral, for example:

$$\text{Dcut}(1 - \text{Hel1}, \text{rank}, \text{num}, \dots),$$

which will not be evaluated if `Hel1` is 1.

3. Parallelization of Helicity Loop

Perhaps the most obvious way to address the OPP slowdown is to parallelize the loop over the helicities. Code generated by FormCalc is in fact well suited for this as FormCalc does not insert explicit helicity states in the algebra already [9]. That is, the amplitude is a numerical function of the helicities λ_i and not a bunch of (different) functions for each helicity combination,

$$\mathcal{M} = \mathcal{M}(\lambda_1, \lambda_2, \dots) \neq \{\mathcal{M}_{--\dots}, \mathcal{M}_{+-\dots}, \mathcal{M}_{-+\dots}, \mathcal{M}_{++\dots}\}$$

In computer science this is known as a Single Instruction Multiple Data (SIMD) design since a single code (\mathcal{M}) is independently run for multiple data (λ_i), and is conceptually easy to parallelize.

Our process model has one master and N workers on an N -core system. The master is in charge of the non-helicity-dependent parts of the computation and coordinates the workers, i.e. starts/stops them and distributes/collects the data. Currently we use the same `fork/wait` technology as in Cuba, with `socketpair` I/O for the data transmission (see Sect. 9).

The implementation is brand new and speed-up measurements are not yet available. Unless the helicity loop accounts for a significant part of the computation time, however, the achievable overall speed-ups may well be limited as the master performs the non-helicity-dependent work single-threaded. Thus, we expect OPP to gain more from parallelization than the PV method.

Unless one is evaluating a single phase-space point only, the helicity parallelization clearly competes for compute cores with Cuba, and it is at present an open question which is the optimal strategy for assigning the cores. Then again, the current `fork/wait` code can be regarded as a proof-of-concept implementation and may be substituted e.g. by a GPU version in the future, for which most of the organizational groundwork is then already laid. For instance, a sizable part of the present effort went into grouping the variables into helicity-dependent and -independent ones, to minimize communication overhead between the master and the workers. Likewise, new versions of the LoopTools functions had to be introduced to obtain control over the cached loop integrals.

The parallelization is enabled in the code by

```
#define PARALLEL
```

which is usually placed in `user.h`. The actual number of cores used can be specified in the environment variable `FCCORES`. If `FCCORES` is not set, all free cores (total cores minus current system load) are used. Note that, at least for now, FormCalc is not aware of the number of cores taken by Cuba, e.g. the `CUBACORES` variable.

4. C Output and Improved code generation

Code generation in C99 is available in FormCalc 7.5 next to the traditional Fortran output. This feature is enabled with

```
SetLanguage[ "C" ]
```

The generated code is binary-compatible with Fortran, i.e. its object files can be linked directly to a Fortran program. The C code accordingly observes Fortran calling conventions (pointers only) and uses underscore-suffixed lowercase function and struct names. One temporary setback is that there is no automatic translation yet of the declarations part of the FormCalc driver modules, e.g. of the model parameters, which are obviously necessary for compiling the C code.

The advantages of C code are threefold:

- It makes integration of generated code into existing C/C++ packages easier (no linking hassles).
- It simplifies GPU programming; for Fortran, only a single, commercial, compiler (PGI) currently targets the GPU.
- One can take advantage of C's `long double` data type which, at least on Intel x86 hardware, gives an additional 2–3 digits of precision at essentially no extra cost. Extended real data types in Fortran, if available, are often IEEE-754-compliant `REAL*16` emulated in software. Only gfortran 4.6+ offers the `REAL*10` Fortran equivalent of `long double`.

Improvements have been made to the Fortran code generation as well:

- Loops and tests are handled through preprocessor macros, e.g.

```
LOOP( var , 1,10,1 )
...
ENDLOOP( var )
```

This aids automated substitution with e.g. `sed`. It also enhances readability (`ENDLOOP` with a variable name rather than an incognito `enddo`) and makes the ‘look’ of the C and Fortran code fairly similar.

- Likewise, the main subroutine `SquaredME.F` is now sectioned by comments. For example, the variable declarations are enclosed in

```
* BEGIN VARDECL
...
* END VARDECL
```

- The generated code and the driver files are consistently formulated in terms of the newly introduced `RealType` and `ComplexType` data types, by default equivalent to `double` precision and `double complex`, respectively. Note that capitalization matters as these words are substituted by the preprocessor. This introduces a level of abstraction which makes it easier to e.g. work with a different precision.

5. Command-line parameters for model initialization

FormCalc includes a suite of so-called driver programs to manage the automatically generated code for computing the squared matrix element. They parse the command line, initialize model constants, set up phase space, etc.

In particular the driver modules for the initialization of the model parameters and luminosity calculation (which includes e.g. the setup of PDFs used in hadronic reactions) had no access to the command-line arguments so far and could use only variables supplied by the user in the main control program `run.F`. In other words, the model inputs and PDF selections were ‘compiled in’ and the executable had to be re-built every time those values changed.

The present command-line parser accepts so-called colon arguments (arguments starting with a ‘:’) before the usual ones on the command line, as in:

```
run :arg1 :arg2 ... uuuuu 0,1000
```

The colon arguments are read into an array (sans colon) and handed to the model-initialization and luminosity-calculation subroutines:

```
subroutine ModelDefaults(argc, argv)

subroutine LumiDefaults(argc, argv)
integer argc
character*128 argv(*)
```

Note that, unlike in C (`char **argv`), fixed-length strings are passed in `argv` since there are no pointers in Fortran 77. It is up to the `ModelDefaults` and `LumiDefaults` subroutines to handle the arguments. In Fortran it is furthermore no fatal error to have no formal arguments in the `ModelDefaults` and `LumiDefaults` subroutines (as in previous FormCalc versions), so old code will compile and run without change.

6. MSSM initialization via FeynHiggs

The colon arguments of the previous section are immediately put to use for the initialization of the MSSM through FeynHiggs [10]. The default MSSM initialization is a stand-alone routine

(i.e. requires no external library to be linked), but is not quite as thorough as FeynHiggs when it comes to the corrections included e.g. in the computation of the Higgs masses.

From FeynHiggs version 2.8.1 on not only the computational engine but the entire Frontend functionality is available through library routines so that the colon arguments can simply be passed to a FeynHiggs subroutine to make FeynHiggs initialize itself as if invoked from its own command-line Frontend. The FormCalc-generated code inherits thus the ability to read parameter files in either native FeynHiggs or SLHA format, and of course obtains all MSSM parameters and Higgs observables from FeynHiggs. There is no duplication of initialization code this way, and moreover the parameters are consistent between the Higgs-mass and the cross-section calculations.

To use the FeynHiggs initialization, one chooses the model-initialization module `model_fh.F` instead of `model_mssm.F`. The compiled code is invoked as

```
run :parafile :flags uuuuu 0,1000
```

The colon arguments are just the ones of the FeynHiggs Frontend: *parafile* is the name of the parameter file and the optional *flags* allows to override the default flags of FeynHiggs.

7. Analytic tensor reduction

Despite the hype that surrounds unitarity methods today, the Passarino–Veltman decomposition of tensor one-loop integrals [7] remains a valuable technique, also because it admits a fully analytic reduction. The complete tensor reduction consists of two steps:

- The Lorentz-covariant decomposition of the tensors of the loop momentum appearing in the numerator into linear combinations of tensors constructed from $g_{\mu\nu}$ and the external momenta with coefficient functions, e.g.

$$\int d^4q \frac{q_\mu q_\nu}{D_0 D_1} \sim B_{\mu\nu} = g_{\mu\nu} B_{00} + p_\mu p_\nu B_{11}.$$

This part has always been performed in FormCalc, as the actual tensors are rather unwieldy objects for further evaluation.

- Solving the linear system that determines the coefficient functions, i.e. expressing the coefficient functions through scalar integrals.

FormCalc has for long included the add-on `FormCalc`btensor`` package which analytically reduces one- and two-point functions when loaded, but higher-point functions could be reduced only indirectly through FeynCalc [11], i.e. the user had to convert/save the amplitudes with `FeynCalcPut`, run FeynCalc in a different Mathematica session, and load the reduced expressions into FormCalc again with `FeynCalcGet`. This procedure was not only suboptimal in terms of user-friendliness but also did not take advantage of the field levels of FeynArts, i.e. FeynCalc always operated on the fully inserted amplitudes rather than the (typically much fewer) Generic amplitudes.

The analytic tensor reduction is meanwhile properly available in FormCalc and can be turned on through the option

```
CalcFeynAmp[... , PaVeReduce → True]
```

Our code implements the reduction formulas of Denner and Dittmaier [12]. While these are fully worked out, it nevertheless took considerable effort to program them in FORM due to at first sight trivial issues, e.g. that there is no straightforward way to obtain the N -th argument of a function. Adding the reduction code to the Mathematica part of FormCalc instead was not an option, however, as we wanted to operate on the Generic amplitudes, before the substitution of the insertions, and this happens in FORM.

Inverse Gram determinants, which appear as a by-product of inverting the coefficient-function system, may lead to instabilities in the numerical evaluation later on and therefore FormCalc tries to cancel them as much as possible. The ones that cannot be cancelled immediately are returned as `IGram[x]` ($= 1/x$) and so can easily be found and processed further in Mathematica.

8. Auxiliary functions for operator matching

As numerical calculations are done mostly using Weyl-spinor chains, there has been a paradigm shift from FormCalc 6 on for Dirac chains, to make them better suited for analytical purposes, e.g. the extraction of Wilson coefficients.

Several `CalcFeynAmp` options allow to arrange Dirac chains in almost any prescribed order so that the coefficient multiplying a product of Dirac chains can be read off easily.

- The `Antisymmetrize` option allows the choice of completely antisymmetrized Dirac chains, i.e. `DiracChain[-1, μ , ν]` = $\frac{1}{2}[\gamma_\mu, \gamma_\nu]$.
- The `FermionOrder` option implements Fierz methods for Dirac chains, allowing the user to force fermion chains into any desired order, e.g. `FermionOrder → {2, 1, 4, 3}` produces fermion chains of the type $\langle 2 | \cdots | 1 \rangle \langle 4 | \cdots | 3 \rangle$. Alternately, `FermionOrder → Colour` brings the spinors into the same order as the external colour indices. If only simplification is sought, `FermionOrder → Automatic` chooses a lexicographical ordering.
- The `Evanescent` option introduces for every application of the Fierz identity a term of the form `Evanescent[original operator, Fierzied operator]` with the help of which one can detect problems due to the application of the Fierz identities.

9. Built-in parallelization in Cuba

Cuba is a library for multidimensional numerical integration which is included in FormCalc but of course can be used independently, too. Only the Mathematica interface was able to compute in parallel so far, by redefining the function `MapSample` with e.g. `ParallelMap`. In the latest release, Cuba 3, we added parallelization also to the C/C++ and Fortran interfaces.

We attempt no parallelization across the network, say via MPI. That is, we restrict ourselves to parallelization on one computer, using operating-system functions only, hence no extra software needs to be installed. A common setup these days, even on laptops, is a single CPU with a number of cores, typically 4 or 8. Utilizing many more compute nodes, as one could potentially do

with MPI, is more of a theoretical option anyway since the speed-ups cannot be expected to grow linearly.

We use `fork/wait` rather than the `pthread*` functions. The latter are slightly more efficient at communicating data between parent and child because they share the same memory space, but for the same reason they also require a reentrant integrand function, and apart from the extra work this takes, a programmer may not even have control over reentrancy in his language, e.g. Fortran’s I/O is typically non-reentrant. `fork` on the other hand creates a completely independent copy of the running process and thus works for any integrand function with almost no restrictions (buffered I/O to a common file, for example, usually leads to unexpected results when executed concurrently).

Because a `fork` is moderately expensive even on Linux with its efficient copy-on-write implementation, we use the ‘spinning threads’ method, i.e. a Cuba routine forks its workers once upon entry and afterwards starts and stops them by sending data or collecting results.

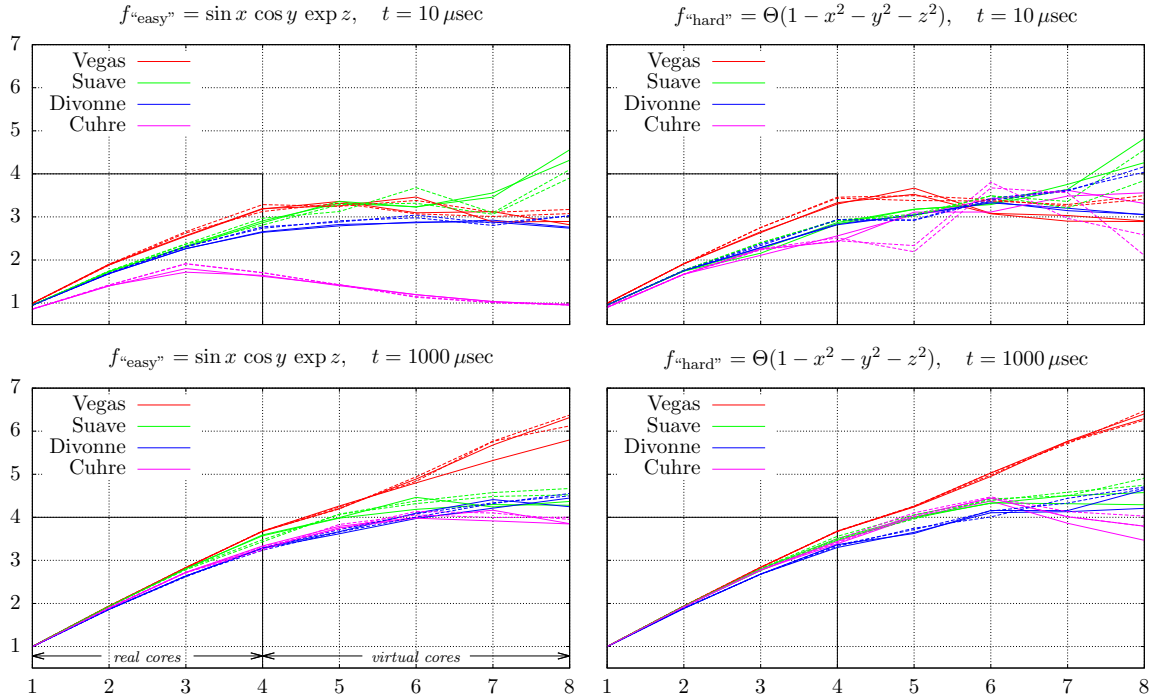


Figure 1: Cuba speed-ups for a three-dimensional integral on an i7-2600 Linux system (3.1.10) with 4 real/8 virtual (hyperthreaded) cores. The vertical line at 4 cores marks the cross-over. The requested accuracy is 10^{-4} in all cases. **Left column:** ‘easy’ integrand. **Right column:** ‘hard’ integrand. **Top row:** ‘fast’ integrand ($10 \mu\text{sec}$). **Bottom row:** ‘slow’ integrand ($1000 \mu\text{sec}$ per evaluation). **Solid line:** shared memory, **Dashed line:** socketpair communication (two curves each to show fluctuations in timing measurements). Note that also in the one-core case a parallel version is used (one master, one worker), which explains why the timings normalized to the serial version are below 1, in the top row visibly so. What appears to be a drastic underperformance of Cuhre in the upper left panel can in fact be attributed to Cuhre’s outstanding efficiency: it delivers a result correct to almost all digits with around 300 samples. In such a case, Cuba may for efficiency choose not to fill all available cores and relative to the full number of cores this shows up as a degradation.

The communication of samples to and from the workers happens through IPC shared mem-

ory (`shmget` and friends), or if that is not available, through a `socketpair`. Remarkably, the former’s anticipated performance advantage turned out to be hardly perceptible. Possibly there are cache coherence issues introduced by several workers writing simultaneously to the same shared-memory area.

Changing the number of cores to use does not require a re-compile, which is particularly useful as the program image should be able to run on several computers (with possibly different numbers of cores) simultaneously. Cuba determines the number of cores from the environment variable `CUBACORES`, or if this is unset, takes the idle cores on the present system (total cores minus load average). That is, unless the user explicitly sets `CUBACORES`, a program calling a Cuba routine will automatically parallelize on the available cores. A master process orchestrates the parallelization but does not count towards the number of cores, e.g. `CUBACORES = 4` means four workers and one master. Very importantly, the samples are generated by the master process only and distributed to the workers, such that random numbers are never used more than once.

Parallelization entails a certain overhead as usual, so the efficiency will depend on the ‘cost’ of an integrand evaluation, i.e. the more ‘expensive’ (time-consuming) it is to sample the integrand, the better the speed-up will be. To give an idea of the values that can be attained, Fig. 1 shows the speed-ups for an ‘easy’ and a ‘hard’ one of the 11 integrands of the demo program included in the Cuba package for two different integrand delays. To tune the ‘cost’ of the integrands, we introduced a calibrated delay loop into the integrand functions (which are simple one-liners and for our purposes ‘infinitely’ fast). The calibration and the timing measurements are rather delicate and shall not be discussed here.

The first, expected, observation is that parallelization is worthwhile only for not-too-fast integrands. This is not a major showstopper, however, as many integrands in particle physics (one-loop cross-sections, for example) safely fall into the 1000- μ sec-and-beyond category.

The second observation is that parallelization works best for ‘simple-minded’ integrators, e.g. Vegas. The ‘intelligent’ algorithms are generally much harder to parallelize because they don’t just do mechanical sampling but take into account intermediate results, make extra checks on the integrand (e.g. try to find extrema), etc. This is particularly true for Divonne, which was originally a recursive algorithm and thus hard to distribute. It took significant effort to un-recurse the algorithm and lift the speed-up curve even this far above 1, but still Divonne is lagging somewhat in parallelization efficiency. Then again, the ‘intelligent’ algorithms are usually faster to start with (i.e. converge with fewer points sampled), which compensates for the lack of parallelizability.

10. Summary

FormCalc 7.5 (<http://feynarts.de/formcalc>) has many new and improved features, most notably OPP methods, C-code generation, the link with FeynHiggs, and analytic tensor reduction. Cuba 3 (<http://feynarts.de/cuba>), included also in FormCalc, parallelizes integrations automatically and achieves decent speed-ups for typical cross-section integrands.

References

- [1] Hahn T, Pérez-Victoria M, 1999, *Comput. Phys. Commun.* **118** 153 [hep-ph/9807565].

- [2] Hahn T, 2001, *Comput. Phys. Commun.* **140** 418 [hep-ph/0012260].
- [3] Hahn T, 2005, *Comput. Phys. Commun.* **168** 78 [hep-ph/0404043].
- [4] Ossola G, Papadopoulos C, Pittau R, 2007, *Nucl. Phys. B* **763** 147 [hep-ph/0609007].
- [5] Ossola G, Papadopoulos C, Pittau R, 2008, *JHEP* **0803** 042 [arXiv:0711.3596].
- [6] Mastrolia P, Ossola G, Reiter T, Tramontano F, 2010, *JHEP* **1008** 080 [arXiv:1006.0710].
- [7] Passarino G, Veltman M, 1979, *Nucl. Phys. B* **160** 151.
- [8] Hahn T, 2010, *PoS ACAT 2010* 078 [arXiv:1006.2231]
- [9] Hahn T, 2003, *Nucl. Phys. Proc. Suppl.* **116** 363 [hep-ph/0210220]
- [10] Frank M, Hahn T, Heinemeyer S, Hollik W, Rzehak H, Weiglein G, 2007, *JHEP* **0702** 047 [hep-ph/0611326].
- [11] Mertig R, Böhm M, Denner A, 1991, *Comput. Phys. Commun.* **64** 345.
- [12] Denner A, Dittmaier S, 2006, *Nucl. Phys. B* **734** 62 [hep-ph/0509141].